

Data Structures

Array and Asymptotic Notation

The Course

● Text Books

- *Introduction to Algorithms* (Third edition)

Cormen, Leiserson, Rivest, and Stein

- An excellent reference you should own

- *Data Structures and Algorithms in C++*

Goodrich, Tamassia, and Mount

● Instructor: Md. Mustaqim Abrar

- Lecturer, CSE Dept, MIST

- *abrar@cse.mist.ac.bd*

What is a Data Structure?

- **Data** is a collection of facts, such as values, numbers, words, measurements, or observations.
- **Structure** means a set of rules that holds the data together.
- A **data structure** is a particular way of storing and organizing data in a computer so that it can be used **efficiently**.
 - Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks.
 - Data structures provide a means to manage huge amount of data efficiently.
 - Usually, efficient data structures are a key to designing efficient algorithms.
 - Data structures can be nested

Types of Data Structures

- Data structures are classified as either
 - Linear (*e.g.*, arrays, linked lists), or
 - Nonlinear (*e.g.*, trees, graphs, etc.)
- A data structure is said to be **linear** if it satisfies the following four conditions
 - There is a unique element called the first
 - There is a unique element called the last
 - Every element, except the last, has a unique successor
 - Every element, except the first, has a unique predecessor
- There are two ways of representing a data structure in memory
 - By means of sequential memory locations (Example: arrays)
 - By means of pointers or links (Example: linked lists)

Arrays

- Data is often available in tabular form
- Tabular data is often represented in arrays
- Matrix is an example of tabular data and is often represented as a 2-dimensional array

Properties of Arrays

- The components of an array are all of the same type.
- Array is a random access data structure.
- Array is a static or variable-length (C99) data structure.
- Access time for an array element is constant, that is, $O(1)$.
- An array is a suitable structure when a small number of insertions and deletions are required.
- An array is a suitable structure when a lot of searching and retrieval are required.

Parameters of Arrays

- **Base Address (b):** The memory address of the first byte of the first array component.
- **Component Length (L):** The memory required to store one component of an array.
- **Upper and Lower Bounds (l_i, u_i):** Each index type has a smallest value and a largest value.
- **Dimension**

Representation of Arrays in Memory

● Array Mapping Function (AMF)

■ AMF converts index value to component address

● Linear (1 D) Arrays:

a : array $[l_1 .. u_1]$ of element_type

Then $\text{addr}(a[i]) = b + (i - l_1) \times L$

$$= c_0 + c_1 \times i$$

Therefore, the time for calculating the address of an element is same for any value of i .

Representation of Arrays in Memory

- **Array Mapping Function (AMF)**

- AMF converts index value to component address

- **2 D Arrays:**

$a : \text{array } [l_1 .. u_1, l_2 .. u_2] \text{ of element_type}$

In which order are the elements stored?

- Row major order (C, C++, Java support it)
- Column major order (Fortran supports it)

Representation of Arrays in Memory

● 2 D Arrays:

The elements of a 2-dimensional array **a** are declared as:

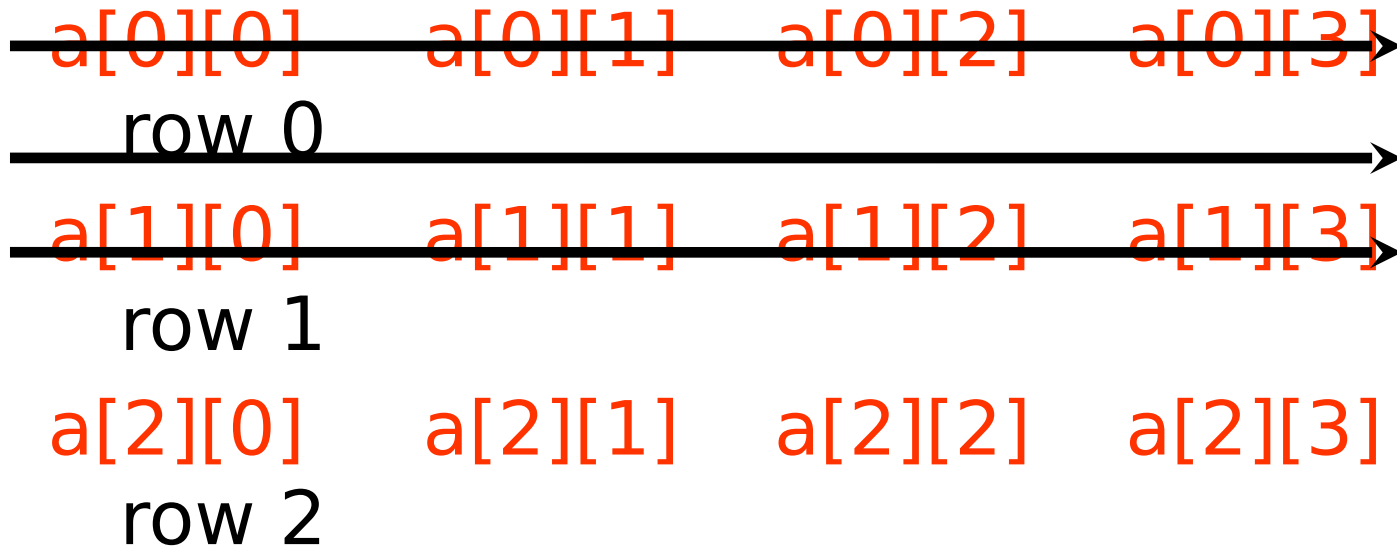
```
int a[3][4];
```

may be shown as a table

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

Representation of Arrays in Memory

Row Major Order:



Representation of Arrays in Memory

● Array Mapping Function (AMF)

■ AMF converts Index value to component address

● 2 D Arrays (Row Major Order):

a : array $[l_1 .. u_1, l_2 .. u_2]$ of element_type

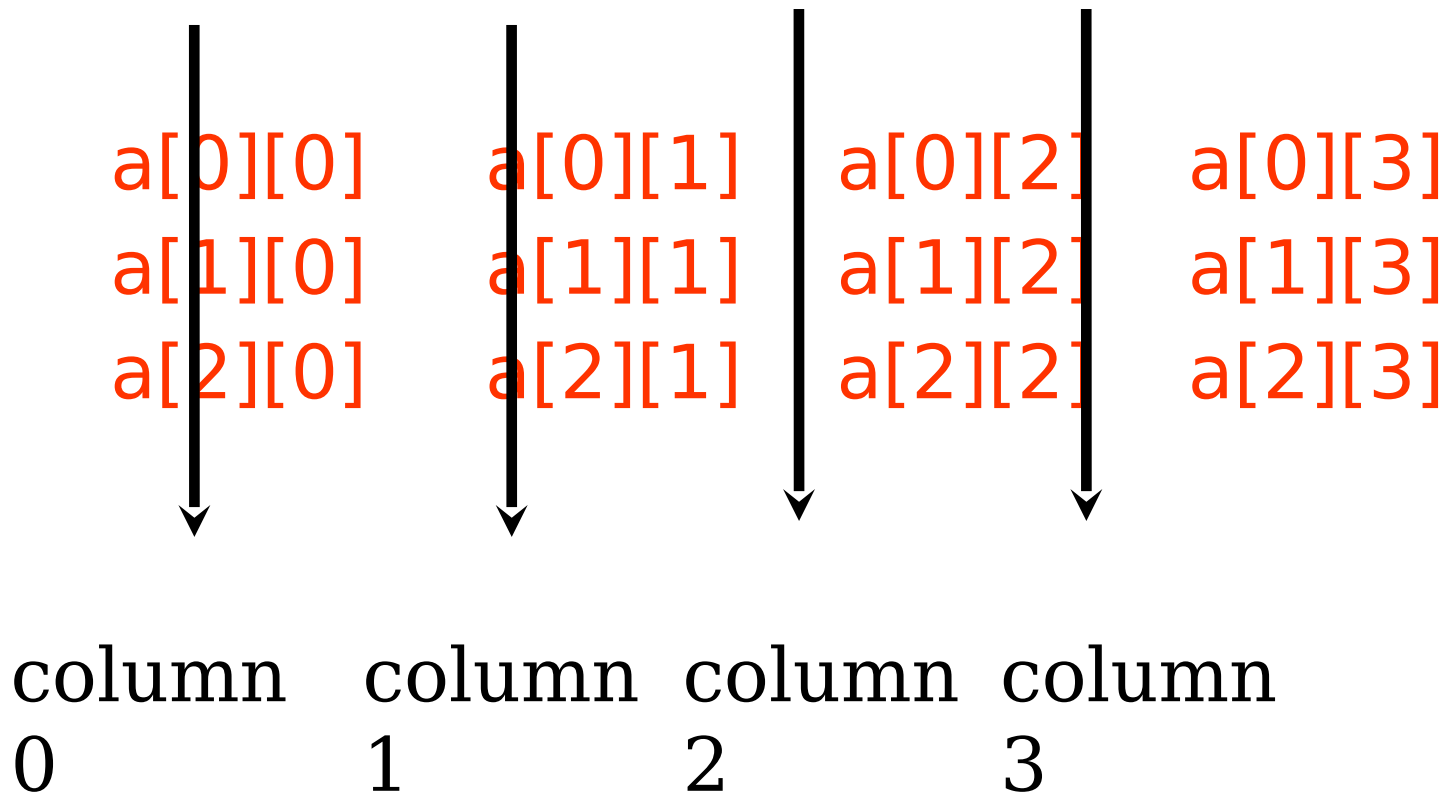
Then $\text{addr}(a[i, j]) = b + (i - l_1) \times (u_2 - l_2 + 1) \times L + (j - l_2) \times L$

$$= c_0 + c_1 \times i + c_2 \times j$$

Therefore, the time for calculating the address of an element is same for any value of (i, j) .

Representation of Arrays in Memory

Column Major Order:



Representation of Arrays in Memory

● Array Mapping Function (AMF)

■ AMF converts Index value to component address

● 2 D Arrays (Column Major Order):

a : array $[l_1 .. u_1, l_2 .. u_2]$ of element_type

Then $\text{addr}(a[i, j]) = b + (j - l_2) \times (u_1 - l_1 + 1) \times L + (i - l_1) \times L$

$$= c_0 + c_1 \times i + c_2 \times j$$

Therefore, the time for calculating the address of an element is same for any value of (i, j) .

Representation of Arrays in Memory

● Array Mapping Function (AMF)

■ AMF converts Index value to component address

● 3 D Arrays :

a : array $[l_1 .. u_1, l_2 .. u_2, l_3 .. u_3]$ of element_type

$$\begin{aligned}\text{Then } \text{addr}(a[i, j, k]) &= b + (i - l_1) \times (u_2 - l_2 + 1) \times (u_3 - l_3 + 1) \times L + \\ &\quad (j - l_2) \times (u_3 - l_3 + 1) \times L + (k - l_3) \times L \\ &= c_0 + c_1 \times i + c_2 \times j + c_3 \times k\end{aligned}$$

Therefore, the time for calculating the address of an element is same for any value of (i, j, k) .

Types of Arrays

- **Static Arrays:** Size is fixed at **compile time** and cannot change.
 - Memory Location: Stack
 - Resizable: No
 - Example: `int arr[5];`
- **Variable-Length Arrays (VLAs):** Size is set at **runtime** based on a variable, but remains fixed once created.
 - Memory Location: Stack
 - Resizable: No (fixed after initialization)
 - Example: `int n = 5; cin >> n; int arr[n];`
- **Dynamic Arrays:** Size is fully **flexible at runtime**, meaning the array can grow or shrink on the fly as you add or remove elements.
 - Memory Location: Heap
 - Resizable: Yes

What is an Algorithm?

- An algorithm is a sequence of computational steps that transform the input into the output.
- An algorithm is a tool for solving a well-specified computational problem.
 - An algorithm is said to be **correct** if, for every input instance, it halts with the correct output
 - An **incorrect** algorithm might not halt at all on some input instances, or it might halt with other than the desired output.

What is a Program?

- A program is the expression of an algorithm in a programming language
- A set of instructions which the computer will follow to solve a problem



Define a Problem, and Solve It

- **Problem:**

- Description of Input-Output relationship

- **Algorithm:**

- A sequence of computational steps that transform the input into the output.

- **Data Structure:**

- An organized method of storing and retrieving data.

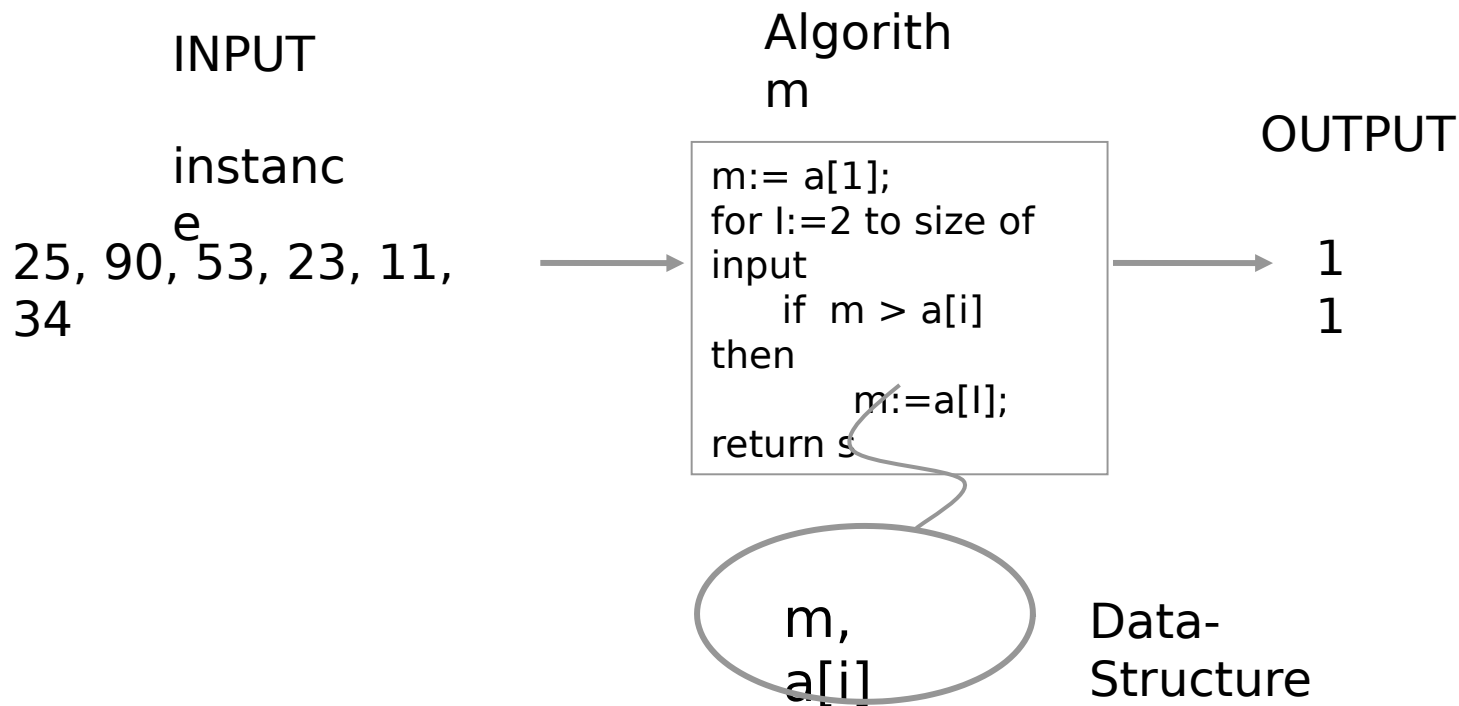
- **Our Task:**

- Given a problem, design a *correct* and *good* algorithm that solves it.

Define a Problem, and Solve It

Problem: Input is a sequence of integers stored in an array.

Output the minimum.



Importance of Analysis of Algorithms

- Need to recognize limitations of various algorithms for solving a problem
- Need to understand relationship between problem size and running time
- Need to learn how to analyze an algorithm's running time without coding it
- Need to learn techniques for writing more efficient code
- Need to recognize bottlenecks in code as well as which parts of code are easiest to optimize

What do we Analyze?

- Correctness

- Does the input/output relation match algorithm requirement?

- Amount of work done (complexity)

- Basic operations to do task

- Amount of space used

- Memory used

- Simplicity, clarity

- Verification and implementation.

- Optimality

- Is it impossible to do better?

Model for Analysis of Algorithms

- Analysis is performed with respect to a computational model
- We usually use a generic uniprocessor random-access machine (RAM)
 - has one processor
 - executes one instruction at a time (no concurrent operations)
 - each instruction takes "unit time"
 - has fixed-size operands, and
 - uses random access memory (RAM)



Input Size

- Time and space complexity
 - This is generally a function of the input size
 - e.g., sorting, multiplication
 - How we characterize input size depends
 - Sorting: number of input items
 - Graph algorithms: number of nodes & edges
 - Etc.

Running Time

- Number of primitive steps that are executed
 - Except for time of executing a function call most statements roughly require the same amount of time
 - $y = m * x + b$
 - $c = 5 / 9 * (t - 32)$
 - $z = f(x) + g(y)$
- We can be more exact if need to be

Running Time

What is the running time of this algorithm (Collatz Conjecture)?

```
PUZZLE( $x$ )  
while  $x \neq 1$   
    if  $x$  is even then  
         $x = x / 2$   
    else  
         $x = 3x + 1$ 
```

Sample run: 7, 22, 11, 34, 17, 52, 26, 13, 40, 20,
10, 5, 16,
8, 4, 2, 1

Asymptotic Performance

- We care most about *asymptotic performance*
 - How does the algorithm behave as the problem size gets very large?
 - **Running time**
 - Memory/storage requirements
 - Bandwidth/power requirements/logic gates/etc.

Asymptotic Analysis

● Worst case

- Provides an upper bound on running time
- An absolute guarantee

● Average case

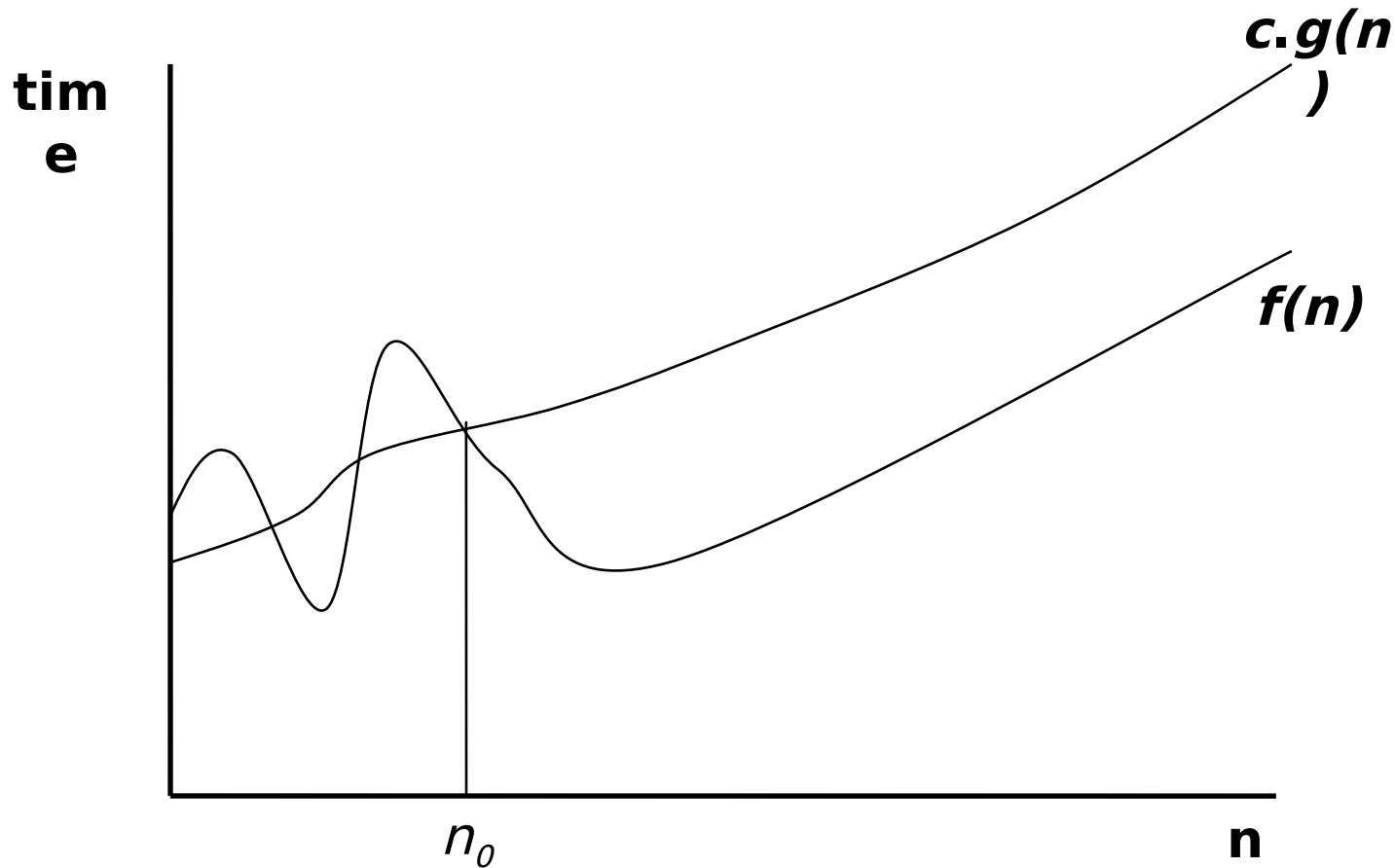
- Provides the expected running time
- Very useful, but treat with care: what is “average”?
 - Random (equally likely) inputs
 - Real-life inputs

● Best case

Upper Bound Notation

- We say Insertion Sort's run time is $O(n^2)$
 - Properly we should say run time is *in* $O(n^2)$
 - Read O as “Big- O ” (you'll also hear it as “order”)
- In general a function
 - $f(n)$ is $O(g(n))$ if there exist positive constants c and n_0 such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$
- Formally
 - $O(g(n)) = \{ f(n) : \exists \text{ positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq c \cdot g(n) \forall n \geq n_0 \}$

Upper Bound Notation



We say $g(n)$ is an *asymptotic upper bound* for $f(n)$

Insertion Sort is $O(n^2)$

●Proof

■ The run-time is $an^2 + bn + c$

○ If any of a , b , and c are less than 0, replace the constant with its absolute value

$$\begin{aligned} \text{■ } an^2 + bn + c &\leq (a + b + c)n^2 + (a + b + c)n \\ &\quad + (a + b + c) \\ &\leq 3(a + b + c)n^2 \text{ for } n \geq 1 \end{aligned}$$

Let $c' = 3(a + b + c)$ and let $n_0 = 1$. Then

$$an^2 + bn + c \leq c' n^2 \text{ for } n \geq 1$$

Thus $an^2 + bn + c = O(n^2)$.

●Question

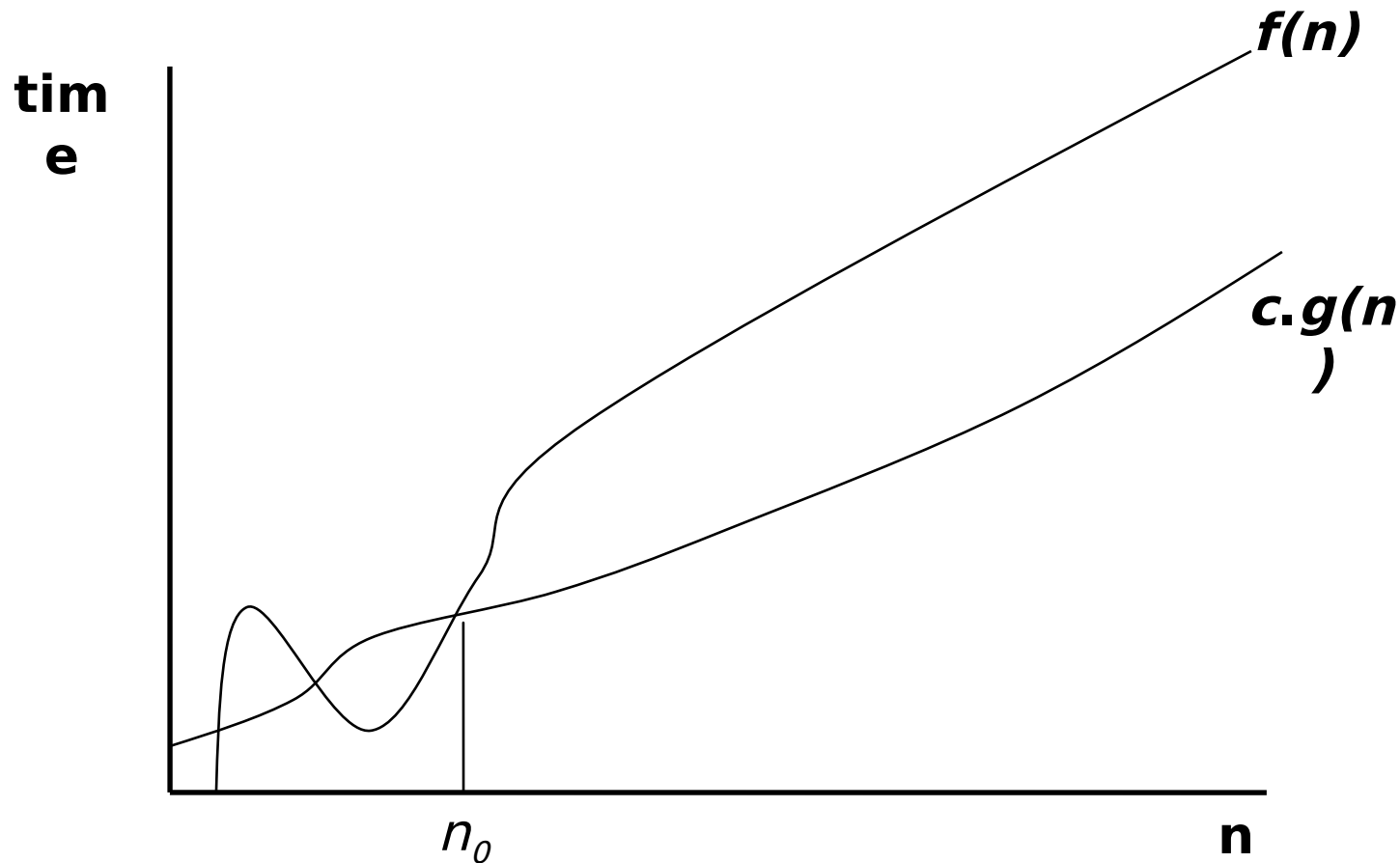
■ Is InsertionSort $O(n^3)$?

■ Is InsertionSort $O(n)$?

Lower Bound Notation

- We say InsertionSort's run time is Big-Omega $\Omega(n)$
- In general a function
 - $f(n)$ is $\Omega(g(n))$ if $\exists$ positive constants c and n_0 such that $0 \leq c \cdot g(n) \leq f(n) \quad \forall n \geq n_0$
- Proof:
 - Suppose run time is $an + b$
 - Assume a and b are positive
 - $an \leq an + b$

Lower Bound Notation



We say $g(n)$ is an *asymptotic lower bound* for $f(n)$

Asymptotic Tight Bound (Big Theta)

- A function $f(n)$ is $\Theta(g(n))$ if $\exists$ positive constants c_1 , c_2 , and n_0 such that

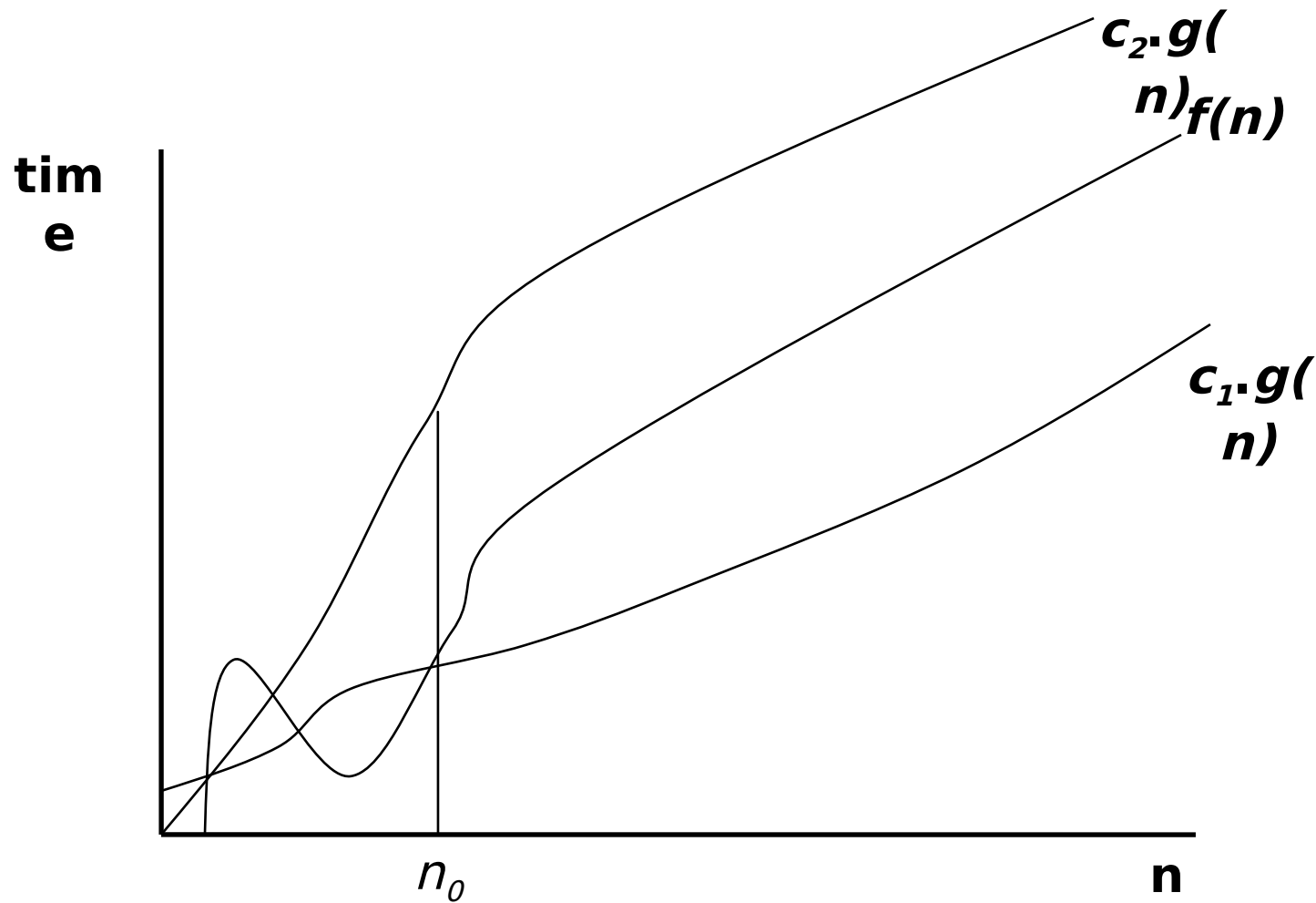
$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall n \geq n_0$$

- Theorem

- $f(n)$ is $\Theta(g(n))$ iff $f(n)$ is both $O(g(n))$ and $\Omega(g(n))$

- Proof:

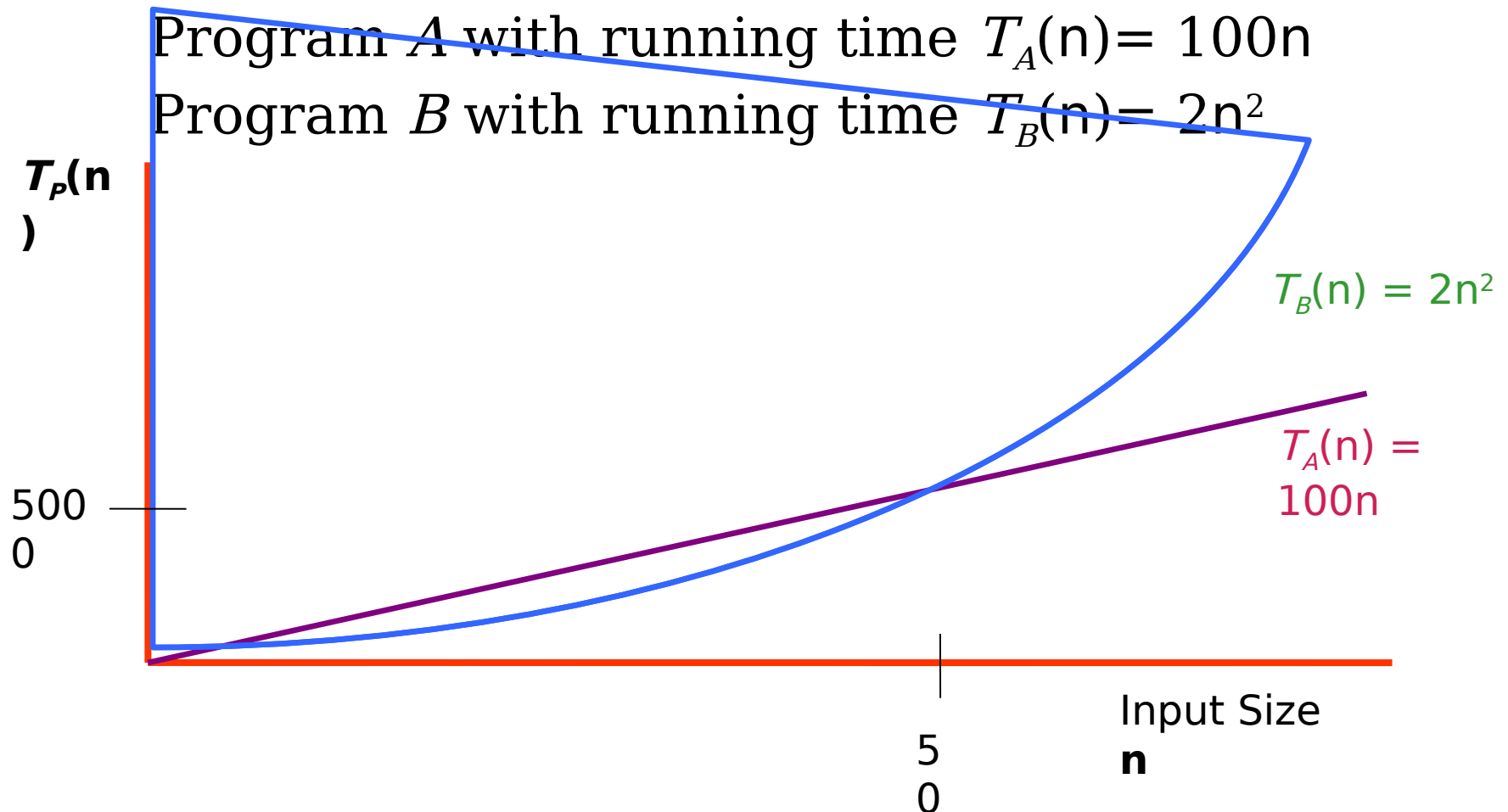
Asymptotic Tight Bound



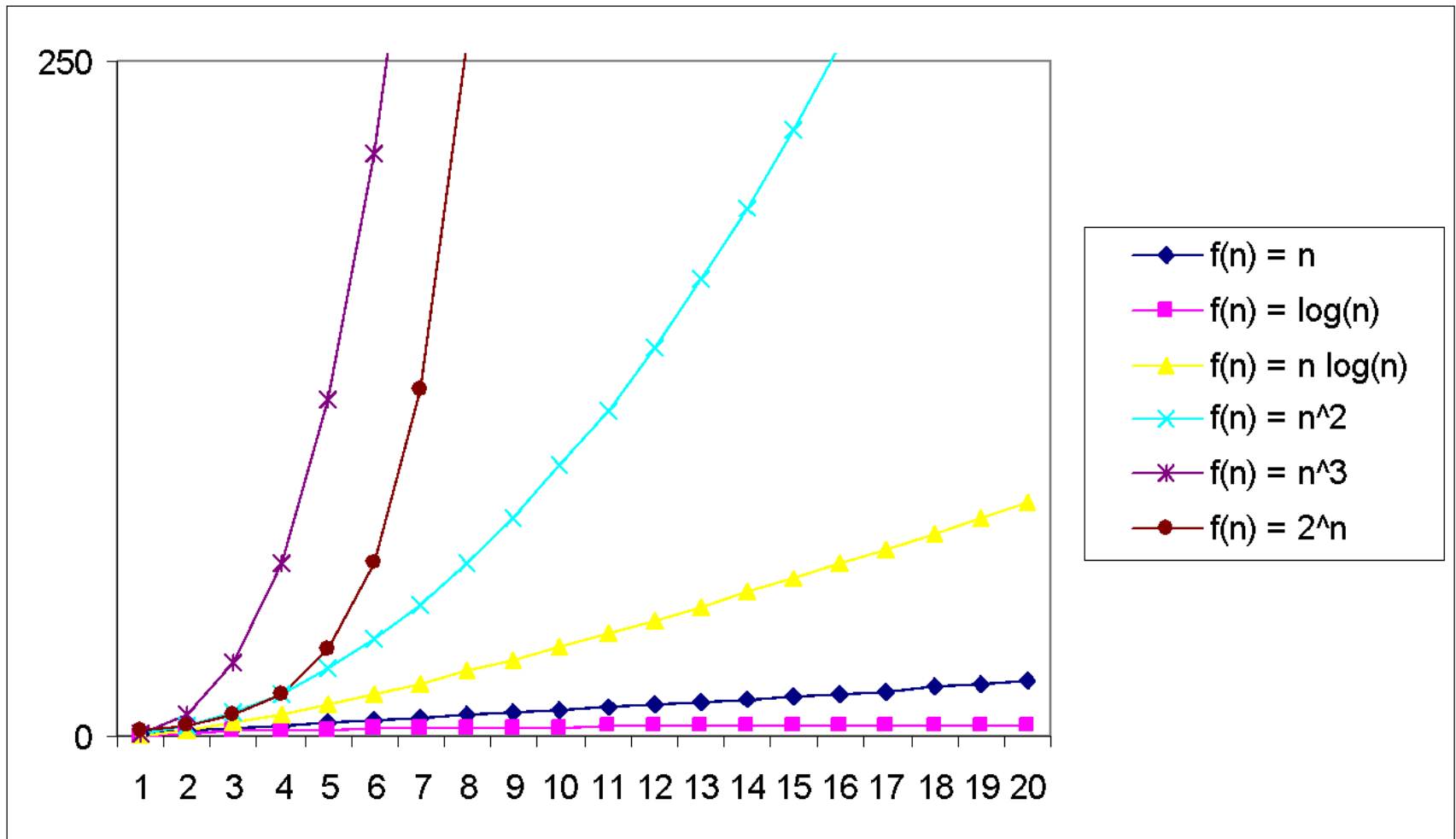
We say $g(n)$ is an *asymptotic tight bound* for $f(n)$

Practical Complexity

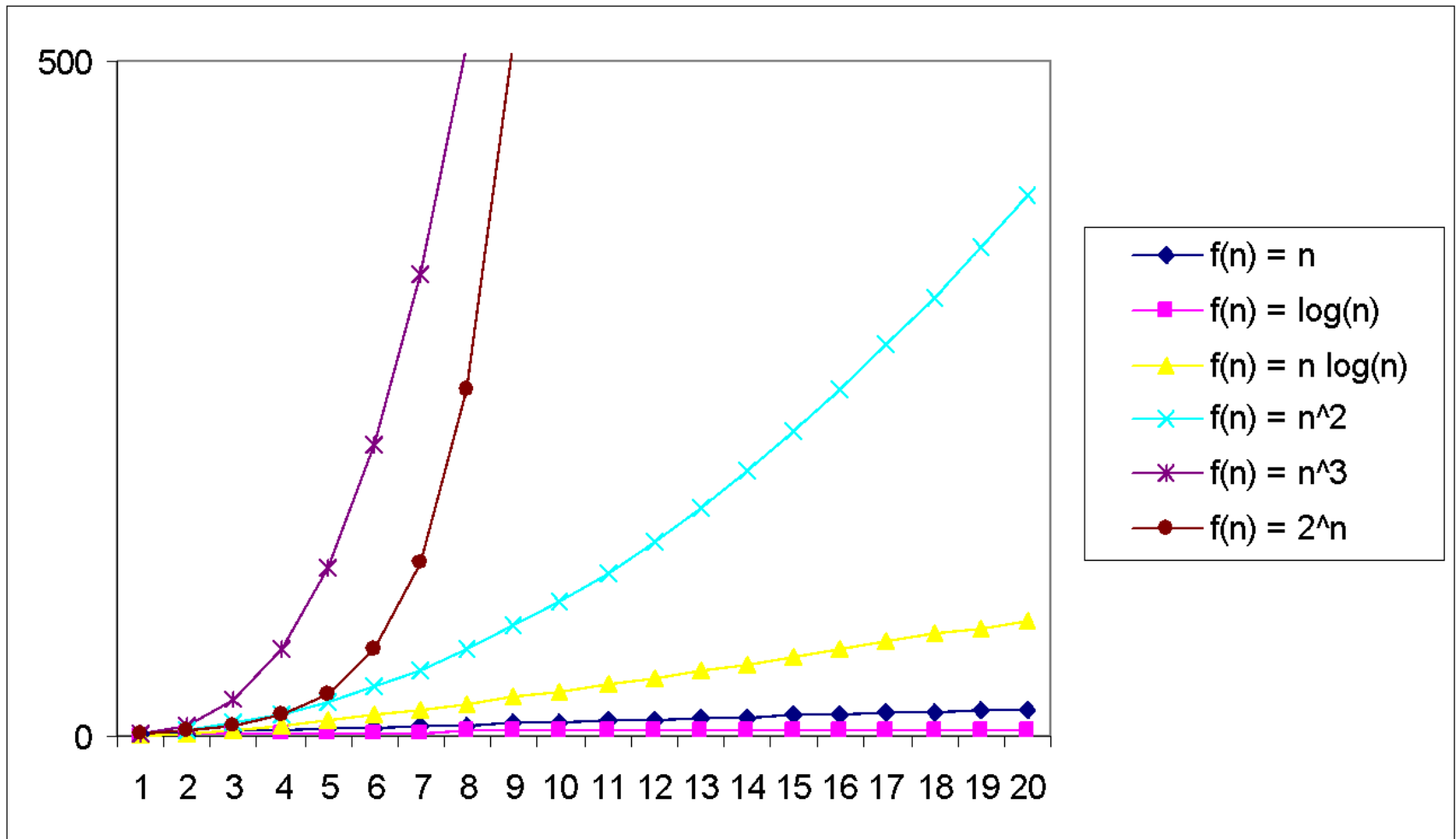
For large input sizes, constant terms are insignificant



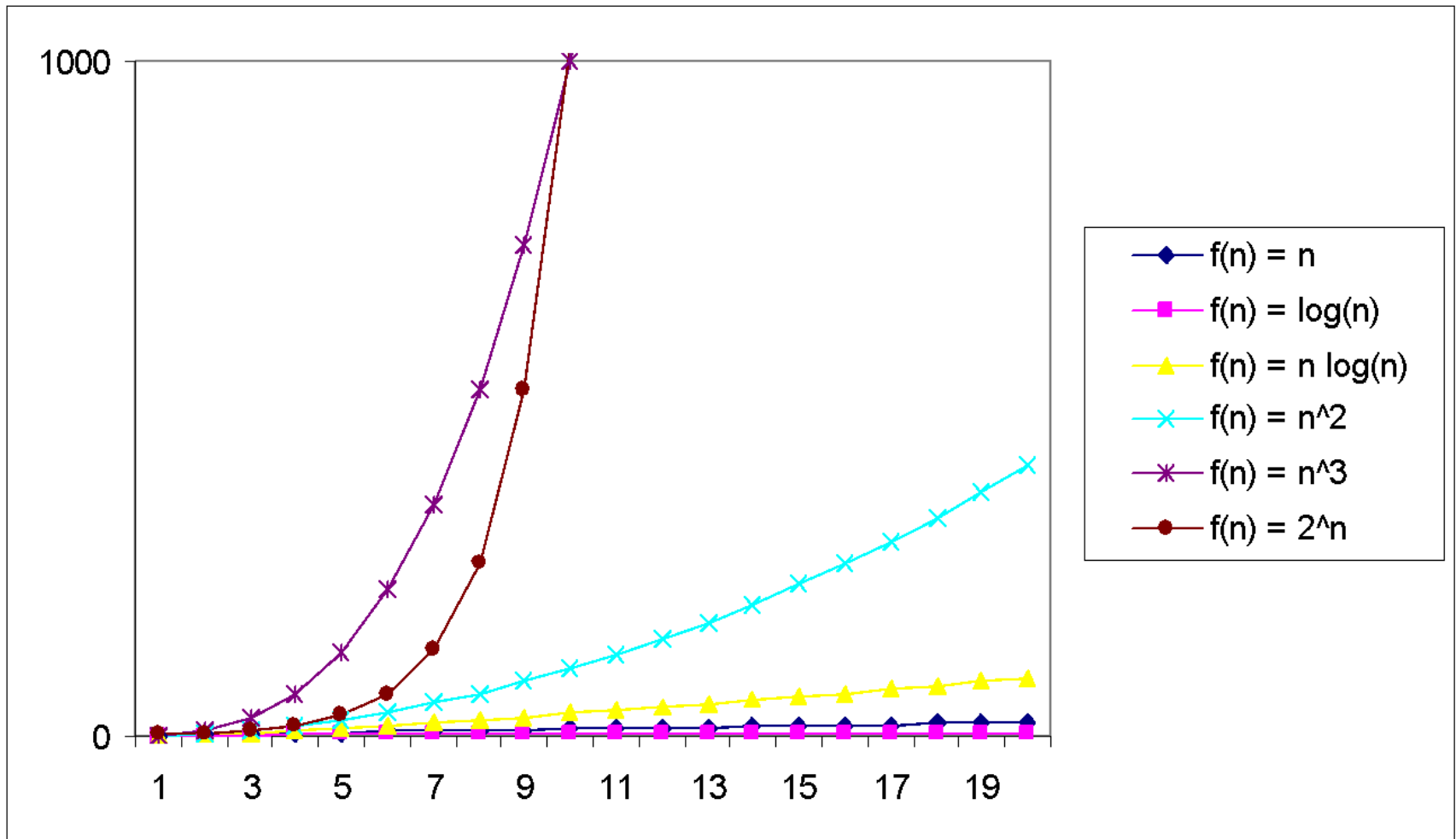
Practical Complexity



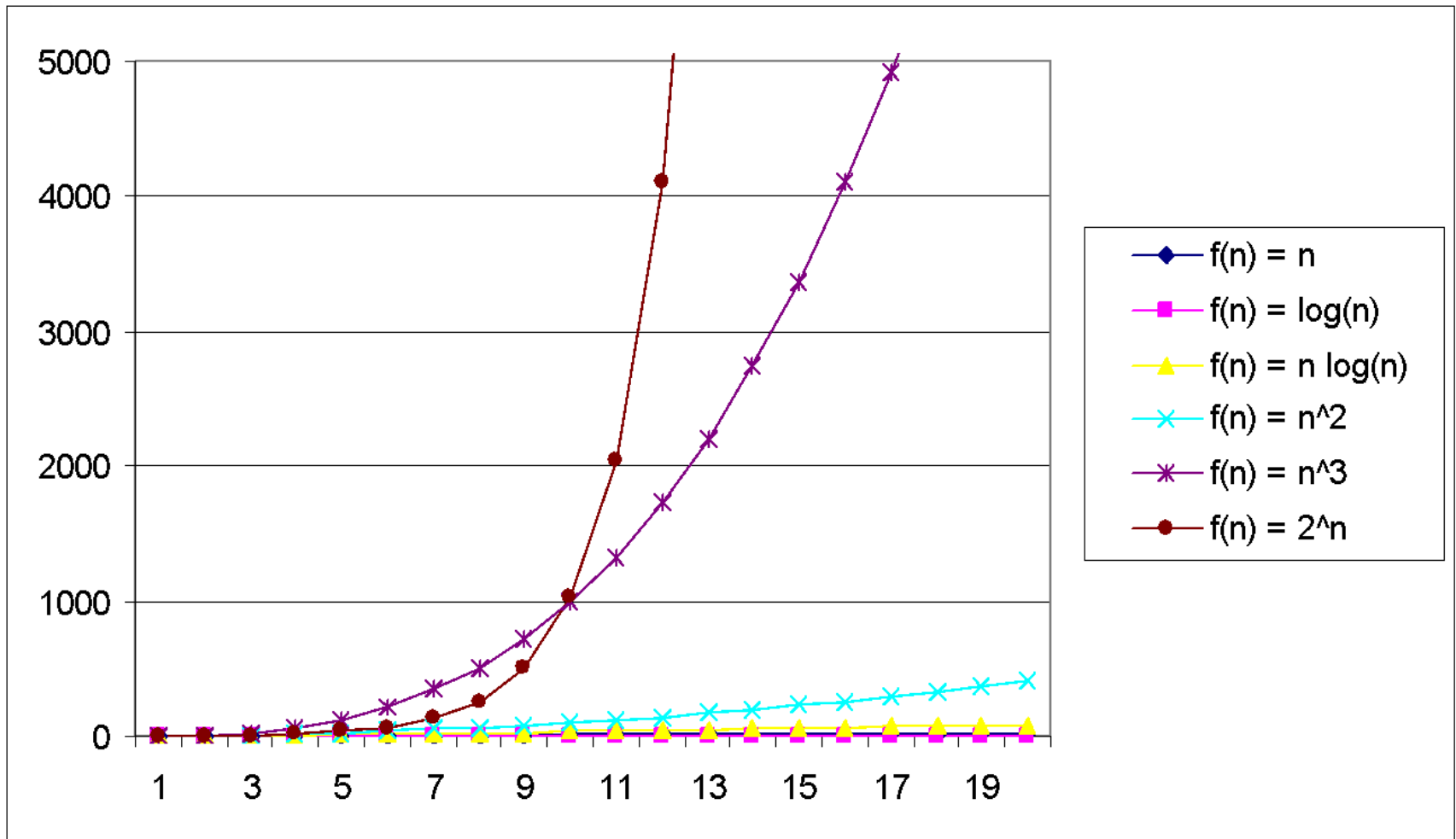
Practical Complexity



Practical Complexity



Practical Complexity



Practical Complexity

Function	Descriptor	Big-Oh
c	Constant	$O(1)$
$\log n$	Logarithmic	$O(\log n)$
n	Linear	$O(n)$
$n \log n$	$n \log n$	$O(n \log n)$
n^2	Quadratic	$O(n^2)$
n^3	Cubic	$O(n^3)$
n^k	Polynomial	$O(n^k)$
2^n	Exponential	$O(2^n)$
$n!$	Factorial	$O(n!)$

Other Asymptotic Notations

- A function $f(n)$ is $o(g(n))$ if $\exists$ positive constants c and n_0 such that

$$f(n) < c g(n) \quad \forall n \geq n_0$$

- A function $f(n)$ is $\omega(g(n))$ if $\exists$ positive constants c and n_0 such that

$$c g(n) < f(n) \quad \forall n \geq n_0$$

- Intuitively,

■ $o()$ is like $<$

■ $\omega()$ is like $>$

■ $\Theta()$ is like $=$

■ $O()$ is like $\leq$

■ $\Omega()$ is like $\geq$

*Thank
you!*